Science of Computer Programming 76 (2011) 1261–1274

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Applying a dynamic threshold to improve cluster detection of LSI☆,☆☆

Pieter van der Spek*, Steven Klusener

Faculty of Sciences, VU University Amsterdam, Amsterdam, The Netherlands

ARTICLE INFO

Article history:
Available online 29 December 2010

Keywords:
Feature extraction
Clustering
Reverse engineering
Software architecture
Latent Semantic Indexing

ABSTRACT

Latent Semantic Indexing (LSI) is a standard approach for extracting and representing the meaning of words in a large set of documents. Recently it has been shown that it is also useful for identifying concerns in source code. The tree cutting strategy plays an important role in obtaining the clusters, which identify the concerns. In this contribution the authors compare two tree cutting strategies: the Dynamic Hybrid cut and the commonly used fixed height threshold. Two case studies have been performed on the source code of Philips Healthcare to compare the results using both approaches. While some of the settings are particular to the Philips-case, the results show that applying a dynamic threshold, implemented by the Dynamic Hybrid cut, is an improvement over the fixed height threshold in the detection of clusters representing relevant concerns. This makes the approach as a whole more usable in practice.

1. Introduction

To developers software often appears as a large number of modules each containing hundreds of lines of code. It is, in general, not obvious which parts of the source code implement a given concern. This fact poses various problems when it comes to maintaining the software. The software records knowledge, expertise, and business rules that may not be available anywhere else than in the source code. Typically, existing documentation is outdated (if it exists at all), the system's original architects are no longer available, or their view is outdated due to changes made by others. So, due to a lack of understanding, maintenance introduces incoherent changes which cause the system's overall structure to degrade [1]. Understanding the system in turn becomes harder any time a change is made to it.

A technique which has been used to provide a view on the system is Latent Semantic Indexing (LSI) [2–4]. This view is used to regain some of the long lost knowledge and aid in the maintenance on the system. LSI is used to identify the associations between words used in source code. The associations are used to cluster the source code documents together. These clusters are interpreted as concerns which reveal the intention of the code [5–9].

The approach as such has been described by various authors [4,6] including the authors of this paper [9], but still a lot of questions remain with respect to choosing various settings. An important aspect of the approach is obtaining the clusters, which provide information on the concerns in the code. The clusters are obtained by having a hierarchical clustering algorithm construct a tree structure called a dendrogram. Subsequently, by cutting the dendrogram the source code documents are grouped into individual clusters.

Usually, the dendrogram is cut using a fixed height threshold. However, there is no such thing as a cutting threshold that works in all circumstances. For instance Kuhn et al. [6] use 0.5, while Maletic et al. [4] use 0.7. The appropriate value,

☆ This work has been carried out as a part of the DARWIN project at Philips Healthcare under the responsibilities of the Embedded Systems Institute. This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

☆☆ Note: Some of the visualizations presented make heavy use of colors. Please obtain a color copy of the article for better understanding.

* Corresponding author.
E-mail addresses: pvdspek@cs.vu.nl (P. van der Spek), steven@cs.vu.nl (S. Klusener).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.12.004

therefore, depends on the particular software system under investigation. The "easiest" way for determining this value is by means of visual inspection of the dendrogram resulting from the hierarchical clustering algorithm, which is difficult when the tree becomes large.

Furthermore, in our experiments at Philips Healthcare it turned out that this threshold results in mostly meaningless clusters as this approach creates one or two disproportionally large clusters and several smaller ones, confirming similar experiences reported in other studies [6,7]. By taking the shape of the dendrogram into account it should be possible to obtain better clusters. This can be done by using a dynamic threshold, which varies in different sections of the tree.

To overcome these deficiencies we have used the Dynamic Hybrid cut [10], which does take the shape of the dendrogram into account. We have performed two case studies on the source code of Philips Healthcare where we applied the approach and compared the outcome of the fixed height threshold to the results obtained by using the Dynamic Hybrid cut.

We make no claims about the generalization of the specific numbers mentioned in this article, but in time they can form part of a broader picture. The method described in this article, however, is more generally applicable and is relevant in other companies and for different software portfolios. In short, the main contributions of this work are as follows.

- We show that the de facto standard for cutting dendrograms, the fixed height cut, is not always the most appropriate choice.
- We evaluate the Dynamic Hybrid cut as an alternative to the de facto standard for cutting dendrograms, the fixed height cut.
- We present the results from two case studies performed at Philips Healthcare comparing the results from the fixed height cut to those of the Dynamic Hybrid cut.

Structure of the paper. In Section 2 we present some related work. Next, in Section 3, we provide an introduction to LSI and its application to source code. Section 4 introduces the Dynamic Hybrid cut, while Section 5 provides the setup for the two case studies. Sections 6 and 7 present the two case studies and their results. Subsequently, in Section 8 we discuss some practical applications of the approach. Finally, in Section 9 we provide some thoughts on possible limitations of the algorithm and our experiments and in Section 10 we present our conclusions.

2. Related work

Many papers report on using LSI for software engineering tasks. Some of the software engineering problems, related to concept location, which have been addressed using LSI are program understanding [4,8,6], high-level concept clones identification [11], conceptual cohesion [12], coupling [13], and traceability (e.g. requirements or documents) [14–16]. Outside the domain of LSI-based approaches, various other approaches have been developed using program slicing [17], dynamic analysis [18], historical information [19], and even simple string pattern matching [20].

A technique which is gaining popularity as an alternative information retrieval technique to LSI is Latent Dirichlet Allocation (LDA) [21–24]. As it does not rely on a hierarchical clustering technique, but rather assigns a probability that a context belongs to a concern, it should be able to identify crosscutting concerns. However, a disadvantage of LDA is that is does not derive any interrelationship between extracted concerns [24,25]. The correlated topic model [26], which builds on LDA, should resolve this issue but has not yet been applied to source code. A hierarchical clustering algorithm does allow for identifying such relationships. Using the Concern Tree they are visualized in an easy-to-understand manner.

Most, if not all, of the approaches using LSI apply clustering and a tree cutting strategy for detecting clusters. An extensive overview of clustering approaches, which are available for software maintenance tasks, is presented in [27]. Clustering approaches have been used in a wide variety of ways to aid software maintenance usually as a means for understanding the (structure of) code. They have been used on their own [28–30] or in combination with other techniques such as formal concept analysis [31,32]. Formal concept analysis on its own turned out to be less suitable for such tasks, especially when applied to large software archives [33]. Furthermore, several researchers have provided methods for evaluating the clusterings made using these algorithms [34,35].

Tree cutting is usually done by means of a single, horizontal cut of the tree structure resulting from the hierarchical clustering. However, often there is no single horizontal cut which accurately reproduces the desired clustering, even when the clusters are easily spotted by means of visual inspection [36]. In this paper, therefore, we have investigated an alternative tree cutting strategy and compared the results to those obtained by using a horizontal cut. Various alternative cutting strategies are available in the literature, such as runt pruning [37], and minimum discrepancy [38]. However, these algorithms use some kind of background knowledge to determine the clusters in the tree. The alternative algorithm evaluated in this paper does not use such knowledge and is therefore ideally suited to be used together with the tree created from the information gathered as part of our approach.

Thus, our work complements existing working using hierarchical clustering as well as our own work presented in [9]. By providing a more flexible tree cutting strategy we overcome the deficiency in existing approaches which use a fixed height threshold even when there is not a single, suitable threshold. The Dynamic Hybrid cut presented in this paper dynamically chooses different thresholds in different parts of the dendrogram resulting in a set of clusters which reflect the structure the dendrogram.

3. Locating concerns in source code using LSI

Latent Semantic Indexing is only part of the entire approach which is divided into the following steps:

1. selection of the input
2. preprocessing and indexing
3. Latent Semantic Indexing
4. computing similarities and **clustering**
5. visualization.

In the remainder of this paragraph we will explain each step. We will limit ourselves to those steps which have proved to be successful in the case of Philips Healthcare. We previously covered most of these steps in more detail [9]. For the remainder of this paper we will be focusing on the clustering step which is discussed in greater detail in Section 4. While some steps, such as visualization, are not required for the approach as such, they are necessary when applying the approach in an industrial setting. They provide a means to convey the information to the people who need to use it and, as such, cannot be ignored. We have implemented this approach in a custom application. We have used this application to test the different cutting strategies described in this article.

3.1. Input selection

The first choice that has to be made is a choice for the input in terms of the level of granularity of the input. Usually, either functions or classes are used depending on the programming language in which the application was written. However, in our case the source code consisted of several different languages supporting different programming paradigms. We have examined various different levels of granularity such as functions and classes. From our experiments we found that the most appropriate level of granularity is the level of functions as this mitigates the problem of multiple concerns being combined in a class or interface. Furthermore, by examining the code at the level of functions, instance variables are only taken into account where they are used. While they are usually declared all together in a dedicated part (at the start) of the source file, we are only interested in the relevant combinations of these variables as they are used in a function.

3.2. Preprocessing and indexing

With the appropriate input, the set of functions extracted from the source code, at hand we proceed to the next step: preprocessing the input. Although variable names are most likely the best choice for using as terms, their vocabulary is not as well-defined as is the case with for instance English words. Their meaning is obscured by abbreviations and various (programmer-specific) word combinations. Therefore, extracting variable names from the source code and using them without modification as input for LSI most likely will not provide us with a good result. Therefore, when applying LSI on source code, usually several preprocessing steps are taken in order to improve the effectiveness of the approach. From our experiments the following steps proved helpful.

- *Splitting variable names into individual words*. In order to do this, various programming styles are taken into account in order to recognize the individual words. Examples are camelcasing and the use of hyphens and underscores. By using the approach a variable name like `NrOfSlices` or `nr_of_slices` will be split into the words `nr`, `of`, and `slices`.
- *Identify compound words*. In addition to the common practice of splitting variable names, we introduce an additional preprocessing step based on our experiences with applying this approach at Philips Healthcare. While even at the most recent working conference on mining software repositories (MSR'09), a popular venue for presenting work on source code analysis, several papers use splitting as a preprocessing technique [39–41], in practice it actually happens quite often that two or more words are best kept together even though based on the use of some convention such as camelcase they should be split. Examples in the case of Philips Healthcare are `DataObject`, `ImageFrame`, and `ResourceManager`. For now, we manually create a list of these words, but we are investigating techniques to automate this step as well.
- *Filtering stop words and programming language keywords*. This list filters out stop words, programming language keywords and variable names consisting of only one or two characters. The latter are usually of the form `i` or `x` and are only used as loop counters or temporary variables.
- *Applying a weighting scheme*. The weighting scheme automatically balances out the influence of very rare and very common words.

Furthermore, we do not use stemming although this is customary to do [4,6]. We found that stemming does not provide additional benefit when analyzing source code, which confirms what others [42,43] have found when using a stemming-algorithm for analyzing natural texts using LSI. When we applied stemming, we ran into several problems such as the mixed use of British and American English and words being taken together such as `present` and `presentation` while this is clearly wrong.

Using the individual words we create an index of the input. This index, as we will discuss next, is represented as a word-by-function matrix in which each cell indicates how often a word occurs in that function. More generally, the matrix is referred to as a term-by-context matrix. We use "term" to refer to the preprocessed words. Similarly, we use "context" to

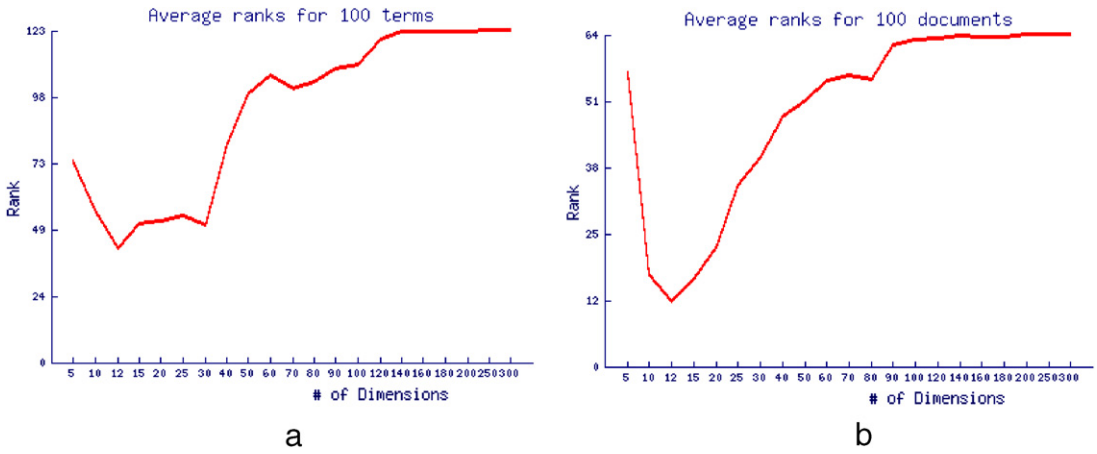


Fig. 1. Average rankings. For each term (context) pair of interest, one term (context) was treated as a query in the space. Vectors for other terms (contexts) were ranked in relationship to this query, based on the cosine measure between those vectors and the query vector. The rank of the paired term (context) in this list was used as the measure of relatedness between the two terms (contexts).

abstract from whether we actually use functions or some other level of granularity for providing the relevant contexts for the terms.

3.3. Latent Semantic Indexing

The matrix resulting from preprocessing and indexing is subsequently processed using Latent Semantic Indexing. LSI is an (almost) fully automated approach and uses no humanly constructed dictionaries, knowledge bases etc. It takes as its input only raw text parsed into terms and separated into meaningful contexts such as sentences or paragraphs, in case of text written in natural languages, or functions or classes, in case of source code written in programming languages.

In the first step of the approach the text is represented as a term-by-context matrix, M, in which each row stands for a unique term and each column stands for a context. In this matrix the $[i, j]$th element indicates the weighted number of occurrences of the ith term in the jth context. This number can be different from the actual number of times a term occurs within a context.

Once we have built our term-by-context matrix, we call upon a powerful technique called Singular Value Decomposition (SVD) to analyze the matrix. The matrix M is decomposed into a matrix describing the original column entries, another matrix describing the original row entries and a diagonal matrix containing scaling values such that when the three components are matrix-multiplied, the original matrix is reconstructed. These special matrices show a breakdown of the original relationships into linearly independent components.

However, many of the elements in the diagonal matrix are very small and may be ignored, leading to an approximation which has many fewer dimensions. Each of the original contexts' similarity behavior is now approximated by its values on this smaller number of factors. Due to this reduction in the number of dimensions, the reconstructed matrix M_k, where k denotes the number of dimensions that have been retained, it is possible for contexts with somewhat different profiles of term usage to be mapped into the same vector of factor values.

The value of k has a profound effect on the effectiveness of the entire approach. An often used rule of thumb is the magic number of 100 dimensions [44], but various other heuristics have been proposed as well [43]. However, as in our case the number of documents is a lot smaller compared to those studies, it seems logical to choose a value lower than that. We therefore use a heuristic suggested by Kuhn et al. [6], which provides good results. This heuristic automatically calculates the appropriate number of dimensions using the following formula: $(m \times n)^{0.2}$ where m represents the number of contexts and n the number of terms.

We have conducted various experiments to evaluate this heuristic. We only present the results here as a description of the complete experiment is outside the scope of this paper. Fig. 1 shows the average rank for 100 pairs of associated terms and contexts, for which we knew that they were correct and thus their ranks should be low, in the context collection at different values of k. The results indicate that the best average performance might be obtained with $k \approx 12$ (see Section 6). As such, these results indicate that values calculated using the heuristic proposed by Kuhn et al. [6] correspond quite well to the optimal number of dimensions and is therefore considered a good rule of thumb when choosing the number of dimensions for applying LSI to source code.

3.4. Computing similarities and clustering

The result of applying LSI is a vector space, based on which we compute the similarity between both contexts and terms. We use this similarity measurement to identify concerns in the source code. To compute the similarity values, an often

used measure is the cosine between the vectors. Although several alternatives exist, the cosine similarity measure works well [3]. Using the similarity measure it is possible to cluster the related contexts. We implemented this in R [45] using the *dist*-function from the proxy package, which actually calculates the dissimilarity. We used this as input for the hierarchical clustering function in R, *hclust*. We use the complete-link clustering algorithm [27]. In order to identify the individual clusters from the tree-structure created by the clustering algorithm we use a tree cutting strategy. We will be going further into how to cut the tree in Section 4.

3.5. Visualization

Several authors [6,46,47] have developed ways to visualize the clusters identified by LSI. These visualizations, however, have several drawbacks. First, although it is possible to use them for larger software systems, they tend to become cluttered in such cases. Furthermore, it is difficult to relate the information to the software system, as these visualizations do not present the identified concerns according the structure of the software system, making it more difficult to discern patterns. Therefore, we have developed an alternative visualization called *Concern Trees*. Its main advantage over existing visualizations is that it maps the clustering onto the familiar structure of the software archive.

The idea behind the concern trees has been derived from the 100% stacked bar chart. This type of chart is used when you have three or more data series and want to compare distributions within categories, and at the same time display differences between the categories. As categories, we have opted for the directories in the directory-hierarchy of the software archive. Each bar represents 100% of the contexts implemented in the source code files of a directory.

For each cluster, we calculate its size in a certain directory by counting the number of contexts which originate from that directory. Thus, if 5 contexts in cluster A originate from directory X which contains 15 contexts in total, the concern tree will show a bar for cluster 1 which occupies one-third of the space. If the remaining 10 contexts are clustered in cluster 2, the remaining two-third of the space will be used to color the bar for cluster 2 in directory X. The bars, therefore, not only represent a cluster, but also its size relative to the other clusters in that directory.

Finally, by mapping the clusters onto the more familiar directory hierarchy together with a legend for each concern, it is easier for the software designer to interpret the results and use them to his advantage. An example is shown in Fig. 3.

4. Dynamic Hybrid cut

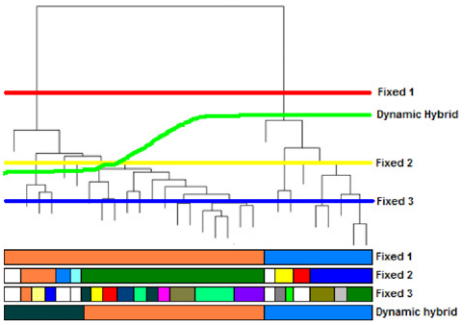
As has been observed by various authors [6,7] LSI often identifies a large domain concern and several smaller concerns. In fact, we observed similar results when applying our approach to the source code at Philips Healthcare (e.g. Fig. 7(a) which shows the fixed height threshold still with several large clusters which were broken down further by the dynamic hybrid cut). Although we do not know whether these observations are the exception or the rule, our findings combined with earlier results in different circumstances, make clear this problem is more general than just the Philips-case and is worth addressing. Although it has been taken for granted until now, we observed that the way in which the clusters are being identified from the dendrogram plays an important role.

As we explained previously, an hierarchical clustering algorithm is used for creating the dendrogram. Hierarchical clustering organizes objects into a dendrogram whose branches are the desired clusters. The process of cluster detection is referred to as tree cutting. The most common, and perhaps naive, way of doing this is by using a fixed height threshold. While distinct clusters may be recognizable by visual inspection, computational cluster definition by a fixed height threshold does not always identify satisfactory clusters. As long as the dendrogram is reasonably symmetric, this will work just fine.

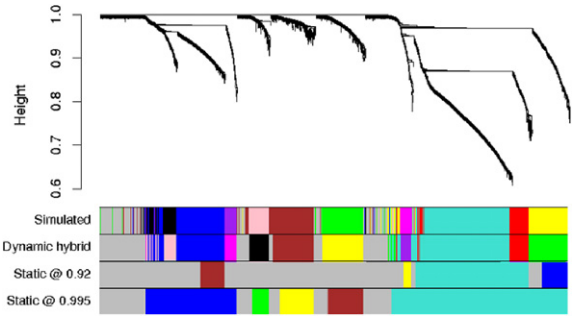
However, in our experiments we found that the tree is often asymmetric. With asymmetric dendrograms the fixed height threshold is not able to identify clusters representing concerns with the same relevance level. It tends to uncover a large domain concern, which in reality consists of several subconcerns. These subconcerns are closely related compared to the generic topics clustered in other parts of the tree. The fixed height threshold is not able to cope with these different relevance levels.

Thus, a fixed height threshold, in this case, results in a few large clusters as shown by the top bar in Fig. 2. Alternatively, a lower threshold results in a lot of singleton clusters and leaves which cannot be clustered at all as they do not meet the threshold, as shown by the lower two bars. Neither result is desirable. In short, with the fixed height threshold, it is not possible to account for the different kinds of concerns, i.e. domain and generic, implemented in the code.

Algorithm description. To overcome this problem, we use a tree cutting method based on analyzing the shape of the branches of a dendrogram. Langfelder et al. [10] have developed a bottom-up algorithm, called the Dynamic Hybrid cut, which we have used as an alternative to the fixed height threshold. The Dynamic Hybrid cut builds clusters in a bottom-to-top manner in two steps. First, the method identifies preliminary clusters as branches that satisfy four criteria: (1) they contain a certain minimum number of objects; (2) objects too far from a cluster are excluded from it even if they belong to the same branch of the dendrogram; (3) each cluster should be separated from its surroundings by a gap; and (4) the core of each cluster should be tightly connected, where by core the lowest-merged objects in the cluster are meant. These criteria are supplied to the algorithm as input parameters by the user. Apart from the first criterion, however, they are best left at their default settings. For the second criterion, this means that it is set to the maximum dissimilarity level so that the entire dendrogram is considered. This has the effect that the algorithm will examine the dendrogram all the way to the top to identify clusters.



(a) Example of different tree cutting strategies. This example is based on a dendrogram build from analysis on source code.



(b) A simulated dendrogram example [10]. The first color band 'Simulated' shows the simulated true cluster membership; color band 'Dynamic Hybrid' shows the results of the proposed tree cutting method; the color band 'Static @ 0.92' shows the results of the standard, constant height cut-off method at height 0.92. The height refers to the *y*-axis of the dendrogram.

Fig. 2. Tree cutting strategies. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Although under certain circumstances this may lead to some detected clusters being larger than optimal – as some unrelated objects near the top of the dendrogram being added to the cluster – it also makes sure that there are no items which get overlooked.

The third and fourth parameter are both derived from the second. In case of the maximum core scatter, this means at most five percent of the maximum dissimilarity. The gap, in that case, is the distance between the maximum core scatter and the maximum dissimilarity. Again, this has the effect that the entire dendrogram is considered thus taking each and every item into consideration for clustering.

Based on these criteria, the algorithm will proceed with merging branches of the dendrogram until it arrives at a point where two branches, selected to be merged together, both satisfy these criteria. In that case, both branches are declared "closed" and no further objects will be added as part of this step. This "branch pruning" step is based on the merging information of the dendrogram (but not on the order of the clustered objects).

The second, optional, step of the algorithm will assign previously unassigned objects (either singleton objects or branches) to the nearest cluster. The algorithm uses the average dissimilarity to determine the cluster to which to assign the unassigned object. In the case that the left-over object is too far away from any other cluster, as defined by the second criterion mentioned above, the object will remain unassigned.

Relevance level. Thus, the only setting which needs to be provided is the desired minimum cluster size as the other setting is best left at its default value. While the appropriate minimum cluster size will differ per software system, the values used in this paper are useful as a reference. By taking the minimum cluster size and the shape of the tree into account, instead of using a fixed height threshold, we treat the different parts of the tree differently. As depicted by the examples in Fig. 2, the Dynamic Hybrid cut cuts the tree at different heights in the two parts of the tree. A simulated example in Fig. 2(b) shows the difference between the actual, simulated clusters. This example was taken from [10] and displays a simulated gene expression data set. This type of data results in more clearly recognizable branches compared to a clustering based on source code. The top bar shows the clusters as generated from the simulation. The second bar shows the clusters as identified by the Dynamic Hybrid cut. Some of the smaller clusters are missing, but overall the detection rate is very good. The last two bars show two different fixed height thresholds. These cuts are not able to deal with different relevance levels in the tree and the detected clusters look nothing like the actual, simulated clusters. Although, as shown in Fig. 4, with dendrograms resulting from source code the differences are not as large, the Dynamic Hybrid cut still is an improvement over the fixed height threshold.

The larger differences in similarity in the right part of the tree are treated differently compared to the left part of the tree. With the fixed height threshold, this would have been impossible. In practice, domain concerns are clustered more closely together compared to generic topics such as logging and user interface handling, due to a greater overlap in the terms being used. The fixed height threshold is not able to cope with this phenomenon. The Dynamic Hybrid cut, however, treats these concerns differently and is therefore able to distinguish more specific domain concerns while still recognizing more loosely coupled generic concerns.

Limitations. The Dynamic Hybrid cut is not without its own drawbacks, however. First, in a sense we substitute the difficult choice for a fixed height threshold with that of a minimum cluster size. Neither of these choices is backed by any kind of heuristic or other method of choosing a 'good' value. Moreover, we have chosen to use the default settings for the other three parameters to the algorithm, but changing these settings might help in obtaining even better results. Although this does not seem like much of an improvement, we consider this to be an acceptable price to pay for improved flexibility. The fixed height threshold, although easy to use, simply lacks the flexibility required for dealing with the asymmetric dendrograms.

A second drawback is that the algorithm for the Dynamic Hybrid cut is more complex compared to the straightforward fixed height threshold. It depends on the number of objects in the dendrogram, the minimum cluster size and, in case the optional step two is performed, on the number of steps required to assign each left-over branch or object to an existing cluster. However, as the algorithm has been applied to dendrograms with several thousands of objects by the original author, we are confident that this will not be a problem when combined in the approach as described in this article.

5. Experimental setup

To test the effectiveness of the Dynamic Hybrid cut, we have performed two separate case studies on the source code of Philips Healthcare. For the description of the case study protocol we use the components of research design as presented by Yin [48]. We will posit the research questions of the study, the unit of analysis, and the criteria for interpreting the findings.

Questions. The goal of these two case studies is to evaluate the Dynamic Hybrid cutting strategy and compare its results to the fixed height threshold cutting strategy. Our primary interest is the relative difference of the Dynamic Hybrid cut compared to the fixed height threshold. The two case studies should thus answer two fundamental questions. First, *how* does the performance of the two cutting strategies differ and, second, *why* do they differ. Our hypothesis is that the Dynamic Hybrid cutting strategy will perform better compared to the fixed height threshold as it is better capable of cutting asymmetric dendrograms.

Unit of analysis. For this purpose, we analyzed two distinct software systems using our approach. We describe both software systems in detail in Sections 6 and 7. We analyzed each software system twice: once using the Dynamic Hybrid cut and once using the fixed height threshold. We presented the results to two experts – one for each software system – and asked them to rate the quality of the clusters. The rating indicates how well a cluster matches an actual concern in that particular software system according to the expert.

We did not inform them which cutting strategy produced which set of clusters. We only explained the two different cutting strategies and their advantages and drawbacks. By not mentioning which set of clusters was produced by which algorithm, we think this did not interfere with their evaluation. In our experience, it even made them more perceptive of any differences and the possible merit of those differences.

Interpretation. In order to rate a cluster we use a 5-point Likert scale [49], where 1 indicates a cluster is completely wrong (i.e. the associated terms have nothing in common and do not relate to any concern in the code) and 5 indicates the cluster is a perfect match to a concern in the code (i.e. the associated terms belong to the same concern and the algorithm indicates the correct places in the archive where it occurs). The other values indicate varying degrees of a partial match (e.g. not all terms are relevant and/or the cluster occurs in places where it should not). We subsequently calculate the average. The higher this value, the better the performance of the clustering algorithm. More detailed evaluation such as an internal evaluation of the clusters (e.g. by identifying specific contexts which should not be part of a cluster), was not part of the evaluation. This was considered too labor-intensive by the experts and thus not feasible from a practical perspective.

Validation. As the first case study was performed on a relatively small application it was possible for the expert to manually generate a set of reference clusters which we could compare to the generated ones. He indicated the name (as a short description), a rough estimate of the relative size and the location of 14 concerns in the package. The resulting Concern Tree is shown in Fig. 3.

It was too labor-intensive to assign every context to a cluster manually. The expert also did not examine the contexts grouped in the generated clusters for the same reason. This means that the expert did not assign any functions to the reference clusters. He only indicated which clusters are present and their location in the software archive. He also provided a rough indication of their size. Thus, it was not possible to automatically calculate how well the clusters, identified by our approach, matched a particular reference cluster for instance by using the overlap in the amount of contexts. We did compute the overlap between clusters generated by the two cutting strategies, which the expert mapped onto the same reference cluster. This number is not a measure for the quality of the clusters, as we cannot compare the generated clusters to the reference clusters. It does, however, provide some insight into the relative difference between the clusters generated by the different cutting strategies. We used the Jaccard similarity coefficient [50] to compute the overlap. The coefficient, J, is given as:

$$J = \frac{M_{11}}{M_{01} + M_{10} + M_{11}}. \tag{1}$$

For two clusters A and B the variables are specified as follows:

- M_{11} represents the total number of contexts the clusters A and B have in common.
- M_{01} represents the total number of contexts present in cluster B, but not in cluster A.
- M_{10} represents the total number of contexts present in cluster A, but not in cluster B.

Furthermore, the reference clustering is useful for validating the generated clusters and gaining insight in how well the two algorithms are capable of detecting all relevant knowledge in the software. For this purpose, in addition to the

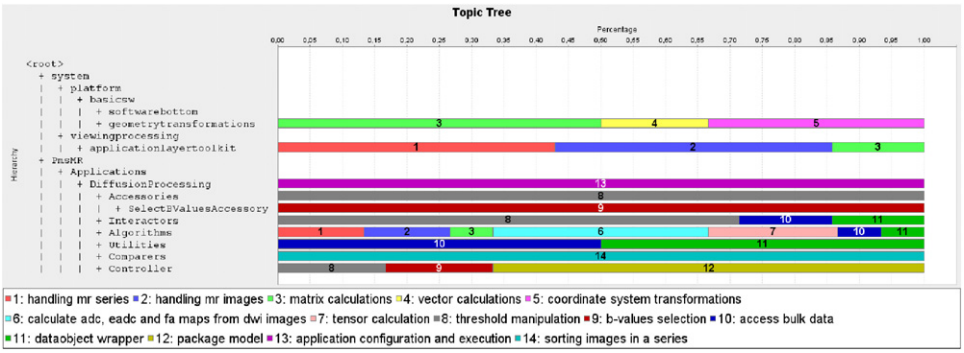


Fig. 3. Concerns in diffusion processing package as identified by the expert.

rating, the expert provided a mapping between the automatically identified clusters onto his own reference concerns. This activity provided us with a list of true positives, false positives, and false negatives. In this case, false positives are clusters which cannot be mapped to an actual concern. False negatives are those concerns identified by the expert, which were not uncovered by our approach. Using these statistics, we calculated precision and recall for both cutting strategies as part of the first case study.

The second case study was already too large for manually listing a set of reference concerns, let alone specifying their distribution. Thus, for the second case study we could not provide insight into any false positives or false negatives. In this case, we only show the relative difference between the two tree cutting strategies.

6. Case study: diffusion processing

6.1. Application description

For the first case study, we have chosen a relatively small clinical application which has been developed by the business unit MRI at Philips Healthcare. The clinical applications constitute a subset of the software and are primarily used for performing post processing operations on previously performed scans. Examples of such applications are diffusion, angiography, spectroscopy and functional MRI.[1] Due to their relatively small size, a single person is responsible for an application. As a contingency, several other persons are available in the event that an immediate repair needs to be made when this expert is not available.

We have chosen an application which is used for diffusion MRI. The diffusion processing clinical application consists of approximately 17.000 SLOC (written in C# and C) and contains various concerns related either directly or indirectly to its main task of computing and displaying the various types of diffusion images.

We have explained some of the general settings already in Section 3. There are also some settings which are specific to the piece of software being analyzed. After preprocessing the input, 407 terms and 819 documents remained. Using the heuristic for calculating the number of dimensions, this results in 12 dimensions to be retained. Furthermore, we have set the minimum cluster size to 32. The choice for this value is based on our knowledge of the system. The smallest set of related functions grouped in a single module consists of 32 functions. Under different circumstances, where this minimum size is not well known, the number can be chosen experimentally. A statistic like the average number of functions in a class is a good starting point. It is reasonable to assume that clusters are not smaller than the average class.

The fixed height threshold was set to 0.9. This value has been chosen such that the resulting number of clusters was almost the same as the number of clusters in the reference set. While this automatically results in less false positives, it might also result in less relevant clusters. A lower threshold, for instance, results in more clusters with potentially a higher coverage of the reference set. In other words, we would be increasing the recall. However, this would come at the cost of precision. The correct clusters would become buried in a pile of nonsense. Such a characteristic is not beneficial for the industrial acceptance of any approach. Thus, our choice was in favor of precision over recall.

6.2. Evaluation results

The two algorithms produce roughly the same amount of clusters. We have chosen the fixed height threshold in such a way that it produced 15 clusters. The Dynamic Hybrid cut resulted in 16 clusters being identified. Fig. 4 shows how the clusters identified by the two algorithms are spread over the dendrogram. Figs. 5 and 6 show the concern trees for both algorithms. It should be noted that the colors are not shared between the different concern trees. Each has its own unique set of colors.

[1] An introduction into these topics can be found here: http://en.wikipedia.org/wiki/MRI.

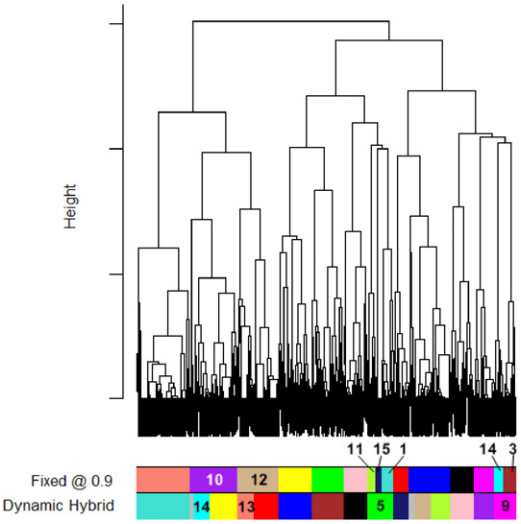


Fig. 4. Hierarchical clustering applied to the diffusion processing package.

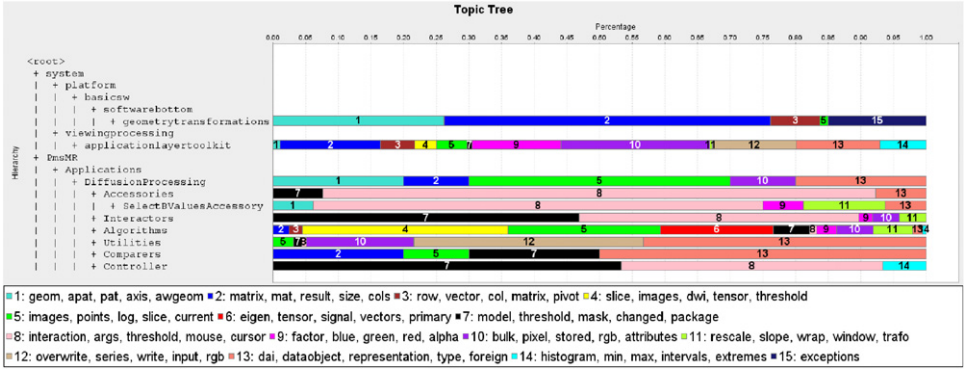


Fig. 5. Concern tree for fixed height threshold.

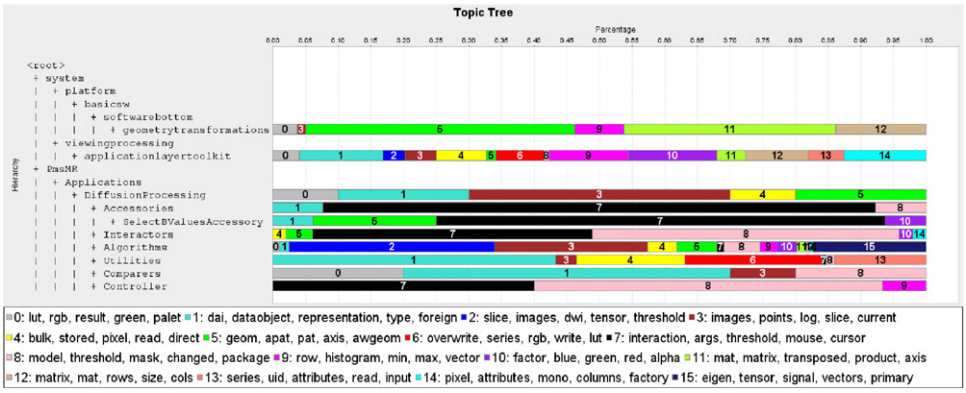


Fig. 6. Concern tree for Dynamic Hybrid cut.

We have performed the evaluation together with the expert on the diffusion processing application. Table 1 shows the scores awarded to the clusters in case of the Dynamic Hybrid cut and fixed height threshold.

We make several observations based on this table. The first and foremost conclusion is that the Dynamic Hybrid cut outperforms the fixed height cut as the average score is 0.7 point higher. The improved performance is also expressed in the higher precision, recall and the resulting F-score. The F-score is calculated as shown in the formula below. The F-score is interpreted as an average of the precision and recall.

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}. \tag{2}$$

Table 1
Likert item scores for both algorithms based on the diffusion case study.

Expert	Dynamic Hybrid		Fixed height		Overlap
	ID	Score	ID	Score	
1	13	3	–	–	–
2	14	4	–	–	–
3	11	4	2	4	0.62
	12	4			
4	9	4	3	4	0.53
5	5	3	1	3	0.48
6	2	5	4	5	1
	3	4	5	4	
7	15	5	6	5	1
8	7	4	7	4	1
	8	4	8	4	
10	1	4	13	3	0.51
	4	4	10	3	
11	6	3	12	3	0.59
Avg. score		3.9		3.2	
False positives		2		4	
False negatives		4		6	
Precision		0.88		0.78	
Recall		0.73		0.65	
F-score		0.80		0.71	

The Dynamic Hybrid cut results in a 10% higher precision and recall. In other words, by using the Dynamic Hybrid cut the expert is presented with more relevant clusters. Also, there are less clusters which do not represent a concern in the source code. In the experience of the expert, the main improvement of the Dynamic Hybrid cut was that it split up some concerns which the fixed height threshold kept together improving the relevance level as explained in Section 4. This difference is also expressed by the numbers showing the overlap. Unfortunately, because we could not compute this number comparing it to the reference clusters, we were not able to identify the reason for the differences in these numbers. We discuss some explicit examples in the coming paragraphs.

The fact that both strategies overlook some concerns has several causes. For instance, cluster 9 in Fig. 3 will never be detected with the current approach, unless steps are taken to improve the approach. The word `b-values` is currently always split into `b` and `values` making it impossible for the approach to detect this concern regardless of the tree cutting strategy being used. Improvements to the approach such as detecting compound words, as explained in Section 3, should resolve this issue.

However, Table 1 shows that with the Dynamic Hybrid cut two additional concerns (clusters 13 and 14) are identified which were not detected using the fixed height threshold. In Fig. 4 we have indicated these two concerns using their cluster number. As depicted there, both of these clusters are a result from the dynamic threshold. Where using the fixed height threshold resulted in two large clusters with relatively low scores, the Dynamic Hybrid cut identifies 5 smaller clusters which mostly received a higher score compared to the scores awarded in case of the fixed height threshold.

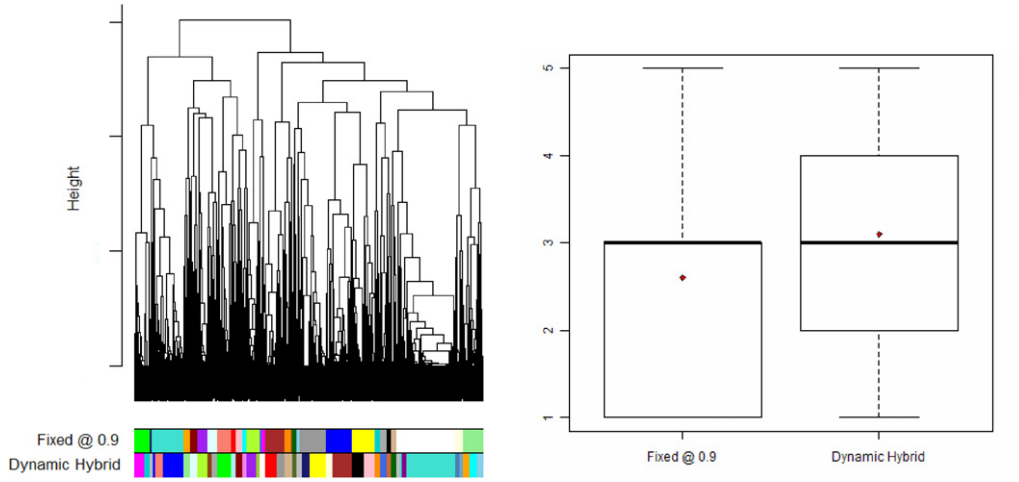
The opposite also occurs. Fig. 4 shows how, in the case of the Dynamic Hybrid cut, clusters 5 and 9 aggregate several contexts, which the fixed height threshold kept separate. In both cases, several clusters which were indicated as false negatives in the case of the fixed height threshold (11, 14 and 15), are now aggregated. The result is a single cluster which is recognized by the expert as a valid concern.

In short, from this case study we conclude that the Dynamic Hybrid cut is an improvement over the de facto standard, the fixed height threshold. It results in considerably less wrongly identified concerns, making it less cumbersome to evaluate the results. Also, the average score for the correctly identified concerns is higher. Thus, the Dynamic Hybrid cut improved the approach by presenting more concerns and more relevant clusters. The case study also shows where the improvement comes from. The dynamically calculated threshold is able to cut the dendrogram at varying heights which results in more relevant clusters.

7. Case study: application development environment

7.1. Application description

For the second case study, we have chosen a coherent part of the software archive developed by the Healthcare Informatics business unit at Philips Healthcare. This business unit is responsible for the maintenance and development of a set of libraries, called Philips Healthcare Informatics Infrastructure (PII). PII is the product family that has been adopted by Philips Healthcare as a software product line engineering approach for medical imaging solutions. It provides a base set of functions which modalities such as MRI, CT and X-Ray can use.



(a) Hierarchical clustering applied to the ADE. (b) Statistics for the ADE evaluation.

Fig. 7. Results from the ADE case study.

We have chosen a single subsystem from the entire software system, called the Application Development Environment (ADE), for our case study. Compared to the diffusion processing application, the ADE is somewhat larger in terms of lines of code. The ADE measures 57.493 SLOC. Mainly this is in C#, but there are also a little over 7000 lines of code written in C++. Some of the parameters are therefore different as well. After preprocessing the source code, 965 terms and 3468 documents were left. Using Kuhn's heuristic for calculating the number of dimensions yields 20 dimensions to be retained. Furthermore, we have set the minimum cluster size to 35. In this case, we did not have sufficient information on the system to determine an accurate value. Thus, we chose a value similar to that in the previous case study. It is slightly higher as the expert expected the minimal set of related functions to be slightly higher due to a difference in programming practices between the PII and MR department. Similar to the previous experiment, the fixed height threshold was set to 0.9. This choice was based on experiments with different settings for the fixed height threshold, where this value produced the best results. While we did not have a set of reference clusters, we knew of a few key concerns that should at least be identified. Using this threshold, resulted in the most hits. Although we have not investigated this any further, this threshold might be more widely applicable at Philips Healthcare.

7.2. Evaluation results

Using the fixed height threshold we found 29 clusters while the Dynamic Hybrid cut produced 38 clusters. Fig. 7(a) shows how the clusters identified by the two algorithms are spread over the dendrogram. As we do not know the actual number of concerns in this software system, we could not calculate precision and recall.

Another concern is the fact that there is a large difference between the retrieved number of clusters. Indeed, in our experiments we found that a different fixed height threshold would result in a number of clusters similar to the number retrieved by the Dynamic Hybrid cut. However, as we already mentioned, this came at the cost of more clusters which did not mean anything to the expert and, even worse, some meaningful clusters were lost. The expert felt that the particular set of clusters resulting from this fixed height threshold resulted in the best outcome compared to other fixed height thresholds we tried. It does give the fixed height threshold a disadvantage compared to the Dynamic Hybrid cut, as it is very likely that this also results in less concerns being identified. However, in our opinion this directly points to the Achilles' Heel of the fixed height threshold: it cannot adapt to variations in the dendrogram and thus cannot keep some clusters together while splitting up other clusters at another similarity level.

The results obtained using these two algorithms already showed much less overlap compared to our findings for the diffusion processing package. The main difference between the two cutting strategies is that the Dynamic Hybrid cut was able to split some clusters into smaller ones without the side effect that a lower threshold has, namely generating additional small, meaningless clusters.

Together with an expert we evaluated the relevance of the clusters generated using both algorithms. As we did not have a set of reference concerns at hand, we only evaluated the relevance of the clusters as such, using the same scale as we used in the previous case study. On average the fixed height threshold scored 2.6 whereas the Dynamic Hybrid cut scored 3.1. However, these numbers only tell part of the story. Fig. 7(b) shows an overview of the scores by means of a boxplot. From this plot it is clear that the distribution of the scores for the clusters identified using the Dynamic Hybrid cut is clearly an improvement compared to the fixed height threshold. This confirms the results of the diffusion processing case study. It shows that the Dynamic Hybrid cut is an improvement over the fixed height threshold.

8. Practical applicability of the results

The results of the ADE case study were not only used for evaluating the effect of a dynamic threshold compared to a fixed threshold. For PII an important goal was to use the results, that is the clusters identified by the approach, to see whether they could be used for improving the documentation on the ADE.

Currently, the documentation on the ADE is written by the engineers responsible for its development. In addition, the support mailing-list is monitored for frequently recurring questions which could indicate a lack of documentation regarding a specific concern. A third method for improving the documentation is that users of the ADE can suggest new concerns. In short, the engineers provide an initial set of documentation and there are two reactive measures in place to continuously improve the existing documentation.

However, the PII organization would also like a proactive way of improving the documentation, i.e. find missing concerns before the users of their software do. As LSI is able to identify the concerns in a particular part of the source code, we used the results of the case study to see whether the clusters identified by LSI could be used to find concerns which were not yet documented.

As the results of the Dynamic Hybrid cut were considered more useful, we used the generated clusters from this approach to construct a list of concerns in the source code. We manually compared those against the list of existing concerns in the documentation. Using this approach, we were able to identify two concerns which were identified by the LSI-approach which were not yet documented. It should be noted that neither of these concerns was part of the list of clusters generated using the fixed height threshold.

So not only did this case study show that the Dynamic Hybrid cut is an improvement in general over the fixed height threshold, but in this particular case we were also able to show a practical use case for the LSI approach as a whole. At the time of writing, the prototype-application used for this study is being reworked by a company to be applied both inside Philips Healthcare and at other companies.

9. Threats to validity

Internal validity. A limitation of these two case studies is that we only had one expert available per case study. It was therefore not possible to compensate for a learning effect. That is, when an expert completed the evaluation of the results from either cutting strategy, his evaluation of the other strategy might be influenced by his prior experience. Also personal bias of the person performing the evaluation plays a role. This could result in an overly negative or positive evaluation. Finally, a lack of sufficient in-depth knowledge on every aspect of the software is of influence on the results. A common way for compensating for such effects is by using multiple persons for the evaluation and presenting them with the two sets of results in different orders, such that the mean score will accurately reflect the difference in performance between the two cutting strategies. However, in practice, application domain knowledge, in most organizations, is thin spread [51]. Also in Philips Healthcare few people combine a broad overview with sufficient in-depth knowledge of the system in order to judge the results from our approach. We compensated for the learning effect and lack of knowledge by allowing the expert to reconsider a prior score. In this way, comparable clusters in both cases received the same score. We countered any personal bias either towards the approach as a whole as well as to the evaluation of the clusters, by explaining in detail the goals of the approach and the evaluation. This makes that the scores are at least comparable and provide an accurate insight into the relative differences between the two strategies.

External validity. Our conclusions are based on two experiments both of which have been conducted at Philips Healthcare. This raises the question whether the conclusions generalize to different software systems and different companies. While the specific settings used in this article are not likely to be useful outside of Philips Healthcare, the approach does not rely on any specific characteristics of the organization Philips Healthcare or its software systems. Moreover, the approach has been applied already on various other occasions as described in Section 2. Finally, as we mentioned in Section 4, others have observed the same limitation of the fixed height threshold as we did. Therefore, it is reasonable to assume that a dynamic threshold is also an improvement in other environments.

Construct validity. A threat to consider is the use of the average Likert score. The average scores are both fractions while the scores that can be awarded are integers. Indeed often the median is used instead of the average to compare groups of Likert scores. However, in our case the median is the same in both cases and thus fails to show the difference between the cutting strategies even though they produce different results. Thus, in our case the average Likert score is more appropriate.

Another possible threat is the clustering algorithm we used. In the two case studies we compared the Dynamic Hybrid cut to the de facto standard, the fixed height threshold. Both algorithms were applied to a dendrogram obtained by using a hierarchical clustering algorithm. Hierarchical clustering is the common approach to clustering used for information retrieval. However, hierarchical clustering has several drawbacks such as sensitivity to noise and outliers, and computational complexity [52]. Flat clustering might be a viable alternative to hierarchical clustering. A well-known flat clustering algorithm is k-means. Flat clustering is efficient and conceptually simple, but it has a number of drawbacks. The algorithms return a flat unstructured set of clusters, require a prespecified number of clusters as input (although algorithms exist to calculate the number of clusters automatically such as X-means [53] or G-means [54]) and are nondeterministic [55].

Hierarchical clustering does not require us to prespecify the number of clusters and most hierarchical algorithms that have been used in information retrieval are deterministic. Furthermore, the dendrogram produced by the hierarchical clustering algorithm holds information on the ordering of the individual objects, and it shows the relations between clusters. This information cannot be obtained from the flat clustering of an algorithm such as *k*-means. While we currently do not use this information, we recognize a potential use for this information as it allows to evaluate the clusters at different levels of abstractions.

Another limitation of the hierarchical clustering algorithm is that each context is assigned to only one cluster. However, in practice a context might belong to multiple clusters as it implements multiple, crosscutting concerns. The current implementation of our approach ignores this possibility and assigns a context to the most prominent concern. However, the approach as presented in this article clusters documents, but it is also possible to cluster the terms. Via the term-by-context matrix the contexts associated with a term can be retrieved together with a probability similar to LDA, which we described in Section 2. Thus, LSI can be modified to behave in a way similar to LDA which is capable of identifying crosscutting concerns. The downside is that it makes the results harder to interpret and convey to the user due to the added layer of abstraction. Therefore, we decided not to cluster the terms. Also, because the users preferred being presented with a single, main concern as opposed to a more nuanced view.

10. Conclusion

In this paper we have shown that the Dynamic Hybrid cut is an improvement over the de facto standard, the fixed height threshold, for cutting the dendrogram. We have performed two case studies at Philips Healthcare in which we compared the Dynamic Hybrid cut to the fixed height threshold. In both cases the Dynamic Hybrid cut outperformed the fixed height threshold. Moreover, in the case of the Application Development Environment of PII these results have successfully been used to improve their documentation of the software.

Overall, we conclude that the Dynamic Hybrid cut improves the effectiveness of the LSI approach for detecting concerns in source code and makes the approach even more usable in practice. The prototype used for the study described in this article, has been transferred to a company to be applied outside of Philips Healthcare as well.

Acknowledgements

We would like to thank Yolanda van Dinther, Linda van Sinten, Amit Ray and Gert Jan Kamstra (Philips Healthcare - PII) for their help in the ADE case study. We would also like to thank Mathijs Visser and Ronald Holthuizen (Philips Healthcare - MRI) with their help in the Diffusion processing experiment. Furthermore, we would like to thank Pierre van de Laar (ESI), Chris Verhoef and Hans van Vliet (VU) for their comments on earlier versions of this paper. Finally, we would like to thank the anonymous reviewers for their valuable comments.

References

[1] M.M. Lehman, Programs, life cycles, and laws of software evolution, Proc. IEEE 68 (9) (1980) 1060–1076.
[2] S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer, R. Harshman, Indexing by latent semantic analysis, J. Amer. Soc. Inform. Sci. 41 (6) (1990) 391–407.
[3] T.K. Landauer, P.W. Foltz, D. Laham, Introduction to latent semantic analysis, Disc. Proc. 25 (1998) 259–284.
[4] J.I. Maletic, A. Marcus, Using latent semantic analysis to identify similarities in source code to support program understanding, in: PICTAI '00, 2000, pp. 46–53.
[5] D. Binkley, D. Lawrie, Information retrieval applications in software maintenance and evolution, in: P. Laplante (Ed.), Encyclopedia of Software Engineering, Taylor & Francis LLC, 2010 (Chapter 2).
[6] A. Kuhn, S. Ducasse, T. Gírba, Semantic clustering: identifying topics in source code, Inf. Softw. Technol. 49 (3) (2007) 230–243.
[7] J.I. Maletic, N. Valluri, Automatic software clustering via latent semantic analysis, in: ASE'99, 1999, p. 251.
[8] J.I. Maletic, A. Marcus, Supporting program comprehension using semantic and structural information, in: ICSE'01, 2001, pp. 103–112.
[9] P. van der Spek, S. Klusener, P. van de Laar, Towards recovering architectural concepts using latent semantic indexing, in: CSMR'08, 2008, pp. 253–257.
[10] P. Langfelder, B. Zhang, S. Horvath, Defining clusters from a hierarchical cluster tree: the Dynamic Tree Cut library for R, Bioinformatics 24 (5) (2007) 719–720.
[11] A. Marcus, J.I. Maletic, Identification of high-level concept clones in source code, in: ASE'01, 2001, p. 107.
[12] A. Marcus, D. Poshyvanyk, The conceptual cohesion of classes, in: ICSM '05, 2005, pp. 133–142.
[13] D. Poshyvanyk, A. Marcus, The conceptual coupling metrics for object-oriented systems, in: ICSM'06, 2006, pp. 469–478.
[14] J.H. Hayes, A. Dekhtyar, S.K. Sundaram, Advancing candidate link generation for requirements tracing: the study of methods, IEEE Trans. Softw. Eng. 32 (1) (2006) 4–19.
[15] G. Antoniol, G. Canfora, G. Casazza, A. de Lucia, E. Merlo, Recovering traceability links between code and documentation, IEEE Trans. Softw. Eng. 28 (10) (2002) 970–983.
[16] A. de Lucia, F. Fasano, R. Oliveto, G. Tortora, Recovering traceability links in software artifact management systems using information retrieval methods, ACM Trans. Softw. Eng. Methodol. 16 (4) (2007) 13.
[17] B. Xu, J. Qian, X. Zhang, Z. Wu, L. Chen, A brief survey of program slicing, SIGSOFT Softw. Eng. Notes 30 (2) (2005) 1–36.
[18] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, V. Rajlich, Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval, IEEE Trans. Softw. Eng. 33 (2007) 420–432.
[19] A. Wierda, E. Dortmans, L. Lou Somers, Using version information in architectural clustering—a case study, in: CSMR'06, 2006, pp. 214–228.
[20] A.V. Aho, Pattern matching in strings, in: Formal Language Theory: Perspectives and Open Problems, Academic Press, New York, NY, USA, 1980 pp. 325–347.
[21] D.M. Blei, A.Y. Ng, M.I. Jordan, Latent Dirichlet allocation, J. Mach. Learn. Res. 3 (2003) 993–1022.
[22] P.F. Baldi, C.V. Lopes, E.J. Linstead, S.K. Bajracharya, A theory of aspects as latent topics, SIGPLAN Not. 43 (10) (2008) 543–562.

[23] Y. Liu, D. Poshyvanyk, R. Ferenc, T. Gyimothy, N. Chrisochoides, Modeling class cohesion as mixtures of latent topics, in: ICSM'09, 2009, pp. 233–242.
[24] G. Maskeri, S. Sarkar, K. Heafield, Mining business topics in source code using latent Dirichlet allocation, in: ISEC'08, 2008, pp. 113–120.
[25] D.M. Blei, J.D. Lafferty, Topic models, in: A. Srivastava, M. Sahami (Eds.), Text Mining: Theory and Applications, Taylor and Francis, 2009.
[26] D.M. Blei, J.D. Lafferty, A correlated topic model of science, Ann. Appl. Statist. 1 (1) (2007) 17–35.
[27] T.C. Lethbridge, N. Anquetil, Approaches to clustering for program comprehension and remodularization, in: Advances in Software Engineering: Topics in Evolution, Comprehension and Evaluation, Springer-Verlag New York, Inc., New York, NY, USA, 2002, pp. 137–157 (Chapter 7).
[28] T.A. Wiggerts, Using clustering algorithms in legacy systems remodularization, in: WCRE'97, 1997, p. 33.
[29] B. Andreopoulos, A. An, V. Tzerpos, X. Wang, Clustering large software systems at multiple layers, Inf. Softw. Technol. 49 (3) (2007) 244–254.
[30] R. Adnan, B. Graaf, A. van Deursen, J. Zonneveld, Using cluster analysis to improve the design of component interfaces, in: ASE'08, 2008, pp. 383–386.
[31] A. van Deursen, T. Kuipers, Identifying objects using cluster and concept analysis, in: ICSE'99, 1999, pp. 246–255.
[32] D. Poshyvanyk, A. Marcus, Combining formal concept analysis with information retrieval for concept location in source code, in: ICPC'07, 2007, pp. 37–48.
[33] M. Glorie, A. Zaidman, A. van Deursen, L. Hofland, Splitting a large software repository for easing future software evolution—an industrial experience report, J. Softw. Maint. Evol. 21 (2) (2009) 113–141.
[34] B.S. Mitchell, S. Mancoridis, Comparing the decompositions produced by software clustering algorithms using similarity measurements, in: ICSM'01, 2001, p. 744.
[35] J. Wu, A.E. Hassan, R.C. Holt, Comparison of clustering algorithms in the context of software evolution, in: ICSM'05, 2005, pp. 525–535.
[36] J.R. Kettenring, The practice of cluster analysis, J. Classification 23 (1) (2006) 3–30.
[37] W. Stuetzle, Estimating the cluster tree of a density by analyzing the minimal spanning tree of a sample, J. Classification 20 (1) (2003) 25–47.
[38] D. Dotan-Cohen, S. Kasif, A.A. Melkman, Seeing the forest for the trees: using the gene ontology to restructure hierarchical clustering, Bioinformatics 25 (14) (2009) 1789–1795.
[39] K. Tian, M. Revelle, D. Poshyvanyk, Using latent Dirichlet allocation for automatic categorization of software, in: MSR'09, 2009, pp. 163–166.
[40] A. Kuhn, Automatic labeling of software components and their evolution using log-likelihood ratio of word frequencies in source code, in: MSR'09, 2009, pp. 175–178.
[41] E. Enslen, E. Hill, L. Pollock, K. Vijay-Shanker, Mining source code to automatically split identifiers for software analysis, in: MSR'09, 2009, pp. 71–80.
[42] S.T. Dumais, J. Nielsen, Automating the assignment of submitted manuscripts to reviewers, in: SIGIR'92, 1992, pp. 233–244.
[43] F. Wild, C. Stahl, G. Stermsek, G. Neumann, Parameters driving effectiveness of automated essay scoring with LSA, in: CAA'05, 2005, pp. 485–494.
[44] S.T. Dumais, Improving the retrieval of information from external sources, Behav. Res. Methods Instrum. Comput. 23 (3) (1991) 229–236.
[45] R Development Core Team, R: A programming environment for data analysis and graphics, R Foundation for Statistical Computing, Vienna, Austria, 2008.
[46] S. Ducasse, T. Girba, A. Kuhn, Distribution map, in: ICSM'06, 2006, pp. 203–212.
[47] X. Xie, D. Poshyvanyk, A. Marcus, 3D visualization for concept location in source code, in: ICSE'06, 2006, pp. 839–842.
[48] R.K. Yin, Case Study Research: Design and Methods, SAGE Publications, 2009.
[49] R. Likert, A technique for the measurement of attitudes, Arch. Psych. 22 (140) (1932) 1–55.
[50] A.K. Jain, R.C. Dubes, Algorithms for Clustering Data, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.
[51] D.B. Walz, J.J. Elam, B. Curtis, Inside a software design team: knowledge acquisition, sharing, and integration, Commun. ACM 36 (10) (1993) 63–77.
[52] R. Xu, D. Wunsch II, Survey of clustering algorithms, IEEE Trans. Neural Netw. 16 (3) (2005) 645–678.
[53] D. Pelleg, A.W. Moore, *X*-means: extending *K*-means with efficient estimation of the number of clusters, in: ICML'00, 2000, pp. 727–734.
[54] G. Hamerly, C. Elkan, Learning the *k* in *K*-means, in: NIPS'03, 2003, pp. 281–288.
[55] C.D. Manning, P. Raghavan, H. Schütze, An Introduction to Information Retrieval, Cambridge University Press Cambridge, England, 2009 (online edition).